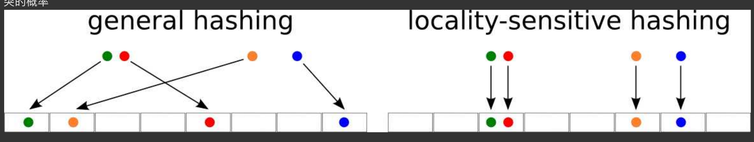
一种实现LSH的最简单的方式是采用random bits sampling的方式，即将待索引的多维整型向量转化为0或1的字符串，
再随机选取其中的K位拼接成新的字符串；最后再采用常规的哈希函数（例如MD5）等算法获取带索引向量的LSH Code。
这样Hamming Distance相近的两个向量，其冲突的概率越大，即结果相等的可能性越大。为了减少增强KNN搜索的能力，采用多个Hash Table增加冲突的概率



局部敏感哈希(Locality Sensitive Hashing，LSH)算法是我在前一段时间找工作时接触到的一种衡量文本相似度的算法。局部敏感哈希是近似最近邻搜索算法中最流行的一种，它有坚实的理论依据并且在高维数据空间中表现优异。它的主要作用就是从海量的数据中挖掘出相似的数据，可以具体应用到文本相似度检测、网页搜索等领域。

回到顶部

1. 基本思想

局部敏感哈希的基本思想类似于一种空间域转换思想，LSH算法基于一个假设，如果两个文本在原有的数据空间是相似的，那么分别经过哈希函数转换以后的它们也具有很高的相似度；相反，如果它们本身是不相似的，那么经过转换后它们应仍不具有相似性。

哈希函数，大家一定都很熟悉，那么什么样的哈希函数可以具有上述的功能呢，可以保持数据转化前后的相似性？当然，答案就是局部敏感哈希。

回到顶部

2. 局部敏感哈希LSH

局部敏感哈希的最大特点就在于保持数据的相似性，我们通过一个反例来具体介绍一下。

假设一个哈希函数为Hash(x) = x%8，那么我们现在有三个数据分别为255、257和1023，我们知道255和257本身在数值上具有很小的差距，也就是说它们在三者中比较相似。我们将上述的三个数据通过Hash函数转换：

Hash(255) = 255%8 = 7;

Hash(257) = 257%8 = 1;

Hash(1023) = 1023%8 = 7;

我们通过上述的转换结果可以看出，本身很相似的255和257在转换以后变得差距很大，而在数值上差很多的255和1023却对应相同的转换结果。从这个例子我们可以看出，上述的Hash函数从数值相似度角度来看，它不是一个局部敏感哈希，因为经过它转换后的数据的相似性丧失了。

我们说局部敏感哈希要求能够保持数据的相似性，那么很多人怀疑这样的哈希函数是否真的存在。我们这样去思考这样一个极端的条件，假设一个局部敏感哈希函数具有10个不同的输出值，而现在我们具有11个完全没有相似度的数据，那么它们经过这个哈希函数必然至少存在两个不相似的数据变为了相似数据。从这个假设中，我们应该意识到局部敏感哈希是**相对的**，而且我们所说的保持数据的相似度不是说保持100%的相似度，而是**保持最大可能的相似度**。

对于局部敏感哈希"保持最大可能的相似度"的这一点，我们也可以从数据降维的角度去考虑。数据对应的维度越高，信息量也就越大，相反，如果数据进行了降维，那么毫无疑问数据所反映的信息必然会有损失。哈希函数从本质上来看就是一直在扮演数据降维的角色。

回到顶部

3. 文档相似度计算

我们通过利用LSH来实现文档的相似度计算这个实例来介绍一下LSH的具体用法。

3.1 Shingling

假设现在有4个网页，我们将它们分别进行Shingling（将待查询的字符串集进行映射，映射到一个集合里，如字符串"abcdeeee"，映射到集合"{a,b,c,d,e}"，注意集合中元素是无重复的，这一步骤就叫做Shingling，意即构建文档中的短字符串集合，即shingle集合。），得到如下的特征矩阵：

shingles \ documents				
	1	0	1	0
	1	1	0	1
	0	1	0	1
	0	0	0	1
	0	0	0	1
	1	1	1	0
	1	0	1	0

其中"1"代表对应位置的Shingles在文档中出现过，"0"则代表没有出现过。

在衡量文档的相似度中，我们有很多的方法去完成，比如利用欧式距离、编辑距离、余弦距离、Jaccard距离等来进行相似度的度量。在这里我们运用Jaccard相似度。接下来我们就要去找一种哈希函数，使得在hash后尽量还能保持这些文档之间的Jaccard相似度，即：

- **Goal:** Find a hash function $h()$ such that:
  - if $sim(C_1,C_2)$ is high, then with high prob. $h(C_1) = h(C_2)$
  - if $sim(C_1,C_2)$ is low, then with high prob. $h(C_1) \neq h(C_2)$

我们的目标就是找到这样一种哈希函数，如果原来文档的Jaccard相似度高，那么它们的hash值相同的概率高，如果原来文档的Jaccard相似度低，那么它们的hash值不相同的概率高，我们称之为Min-hashing(最小哈希)。

3.2 Min-hashing

Min-hashing定义为：特征矩阵按行进行一个随机的排列后，第一个列值为1的行的行号。举例说明如下，假设之前的特征矩阵按行进行的一个随机排列如下：

元素	S1	S2	S3	S4
他	0	0	1	0
成功	0	0	1	1
我	1	0	0	0
减肥	1	0	1	1
要	0	1	0	1

最小哈希值：h(S1)=3，h(S2)=5，h(S3)=1，h(S4)=2.

为什么定义最小hash？事实上，两列的最小hash值就是这两列的Jaccard相似度的一个估计，换句话说，两列最小hash值同等的概率与其相似度相等，即P(h(Si)=h(Sj)) = sim(Si,Sj)。为什么会相等？我们考虑Si和Sj这两列，它们所在的行的所有可能结果可以分成如下三类：

（1）A类：两列的值都为1；

（2）B类：其中一列的值为0，另一列的值为1；

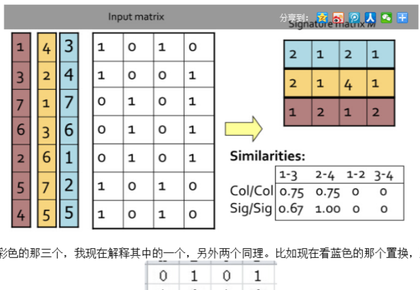
（3）C类：两列的值都为0.

特征矩阵相当稀疏，导致大部分的行都属于C类，但只有A、B类行的决定sim(Si,Sj)，假定A类行有a个，B类行有b个，那么sim(si,sj)=a/(a+b)。现在我们只需要证明对矩阵行进行随机排列，两个的最小hash值相等的概率P(h(Si)=h(Sj))=a/(a+b)，如果我们把C类行都删掉，那么第一行不是A类行就是B类行，如果第一行是A类行那么h(Si)=h(Sj)，因此P(h(Si)=h(Sj))=P(删掉C类行后，第一行为A类)=A类行的数目/所有行的数目=a/(a+b)，这就是最小hash的神奇之处。

Min-hashing的具体做法可以根据如下进行表述：

- Imagine the rows of the boolean matrix permuted under random permutation π
- Define a "hash" function $h_\pi(C)$ = the number of the first (in the permuted order π) row in which column C has value 1:
  $h_\pi(C) = min\ \pi(C)$

返回到我们的实例，我们首先生成一堆随机置换，把特征矩阵的每一行进行置换，然后hash function就定义为把一个列C hash成一个这样的值，就是在置换后的列C上，第一个值为1的行的行号。如下图所示：



图中展示了三个置换，就是彩色的那三个，我现在解释其中的一个，另外两个同理。比如现在看蓝色的那个置换，置换后的Signature Matrix为：

0	1	0	1
1	0	1	0
1	0	1	0
1	0	0	1
1	0	1	0
0	1	0	1
0	1	0	1

然后看第一列的第一个是1的行是第几行，是第2行，同理再看二三四列，分别是1，2，1。因此这四列（四个document）在这个置换下，被哈希成了2，1，2，1，就是右图中的蓝色部分，也就相当于每个document现在是1维。再通过另外两个置换然后再hash，又得到右边的另外两行，于是最终结果是每个document从7维降到了3维。我们来看看降维后的相似度情况，就是右下角那个表，给出了降维后的document两两之间的相似性。可以看出，还是挺准确的，回想一下刚刚说的，希望原来documents的Jaccard相似度高，那么它们的hash值相同的概率高，如果原来documents的Jaccard相似度低，那么它们的hash值不相同的概率高，如何进行概率上的保证？Min-Hashing有个很人的性质：

- Choose a random permutation π
- then $\Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$

就是说，对于两个document，**在Min-Hashing方法中，它们hash值相等的概率等于它们降维前的Jaccard相似度。**

注：在上述例子中，我们还可以采取欧氏距离等相似度量来替代Jaccard相似度，这时候LSH相应的策略也需要进行改变，从而使得最后的hash值等于降为前的相似度。